David Holladay
401 Elmside Blvd.
Madison, Wi 53704
(608) 257-8833 (work)
(608) 241-2498 (home)

Solving Polynominal Equations of Degree Greater than Four with an HP-28S Advanced Scientific Calculator

I have an HP-28S calculator, and a copy of the "Mathematical Applications" Step by Step Solutions For Your HP Calculator. A large section of the book deals with solutions to polynomials. The book contains closed form solutions to polynomials up to the fourth degree. For polynomials of fifth degree and above, there is no closed form solution. I was determined to write a program which would yield all the roots (real and complex) of an arbitrary polynomial using the Graeffe root square method.

This program starts where the PROOT program leaves off. It gives complete solutions (all the roots, real and complex) for polynomials of degree 5 and above. It does so without much manual intervention. The program beeps when it is finished.

There are three drawbacks to the root squaring process on the HP-28S. One is memory. These programs take up about 8k out of the 32k on the machine. The second drawback is that it is slow. It does take several minutes to get a solution.

Finally, the root square method uses big numbers. It would easily bust the 10^{500} limit of the HP-28S. I have a routine that stops the root squaring process before any coefficient reaches 10^{100} Many times, not enough root squaring iterations are done to properly separate the roots. The result is error. To cope with these errors, I have a scoring system to allow the user to instantly know it the results can be relied on or not. If the score shows there are errors, I have a backup routine which separates the good roots from the bad, calculates the reduced equation and tries again. In all cases I have tested, one gets accurate roots in the first or second try. I think that getting all 20 roots (all complex) of a 20 degree polynomial in 45 minutes is acceptable performance.

The operation of the program is simple. Place a polynomial in list form at level 1 of the stack. Execute program MROOT (for master root). After execution, level 1 has a "score" to indicate the accuracy of the roots (a score of 1 is ideal). The second level has the roots in list form. If the polynomial is a linear, quadratic, cubic, or quartic, then software from program PROOT is executed. PROOT is documented in the HP "Mathematical Applications" Step by Step book.

The Approach

The name for the algorithm in this program is the Graeffe root-squaring method with the Brodetsky and Smeal improvement using the Newton-Raphson method for improving the roots. To tell the difference between real and complex roots, the program tests the discriminant b^2-4ac of adjacent trios of coefficients of the final root-squared equation. The root-squaring process goes for 12 steps or until one coefficient reaches 10^{100} (which ever comes first). This program uses the Brodetsky-Smeal improvement for every calcuation.

The book that I have relied on is Numerical Mathematical Analysis by James B. Scarborough, Sixth Edition (John Hopkins Press, 1966). To quote from the book:

"The underlying principle of Graeffe's method is this: The given equation is transformed into another whose roots are high powers of those of the original equation. The roots of the transformed equation are widely separated, and because of this fact are easily found. For example, if two of the roots of the original equation are 3 and 2, the corresponding roots of the transformed equation are 3^q and 2^q, where q is the power to which the roots of the given equation have been raised. Thus if q = 64, we have $3^{64} = 10^{30.536}$, $2^{64} = 10^{19.266}$. The two roots of the given equation were of the same magnitude, but in the transformed equation the larger root is more than a hundred billion times as large as the smaller one. Stated otherwise, that ratio of the roots in the given equation is 2/3, but in the transformed equation it is $10^{30.536}/10^{19.266} = 1/10^{11.27}$, or < 0.00000000001. The smaller root in the transformed equation is therefore negligible in comparison with the larger one. The roots of the transformed equation are said to be separated when the ratio of any root to the next larger is neglible in comparison with unity."

Once a polynomial equation is separated, the roots can be found by taking the qth root of the ratios of adjacent coefficients of the transformed equation. The mechanics of the Graeffe method is to transform the equation so the roots of the new equation are the squares of the previous equation. The process is repeated several times to obtain the desired separation. To separate 2 and 3 as above, the root squaring process would have to be repeated 6 times ($2^6 = 64$).

Again, to quote from Numerical Mathematical Analysis:

"The Graeffe method as explained and illustrated up to this point is sufficient for finding the real roots and one or two pairs of complex roots of an algebraic equation. When an equation has three pairs of complex roots, they can be found without much difficulty by making further use of the relations between roots and coefficients; but since the real parts must be found from a quadratic equation, thus giving two values for the real parts, the proper value must be determined by trial. When the given equation has four or more pairs of complex roots, the practical difficulties in finding them are almost insurmountable.

"Brodetsky and Smeal avoided all these difficulties by moving the origin of x a small distance h and then applying the root-squaring process to the transformed equation. Their procedure enables all roots to be found without any ambiguities and without much additional labor after the roots of the transformed equation has been separated. The Brodetsky and Smeal improvement enables any number of pairs of complex roots to be found with the same ease as one or two pairs. The introduction of the auxiliary variable h more than doubles the labor of separating the roots, but does not cause any other additional labor in the solution."

The Root Squaring Process

If we have a given equation of degree 4, then there are 5 terms. We will write the coefficients as a[1] through a[5] to write the equation as:
$a[1]x^4 + a[2]x^3 + a[3]x^2 + a[4]x + a[5] = 0$

To manually perform one step of the root squaring process, take a polynomial and place the even terms on the left side and the odd terms on the right as follows:

$a[1]x^4 + a[3]x^2 + a[5] + = -a[2]x^3 - a[4]x$

square both sides and substitute $y = -x^2$ you get:

$$a[1]^2y^4 + (a[2]^2 - 2a[1]a[3])y^3 + (a[3]^2 - 2a[2]a[4] + 2a[1]a[5])y^2 + (a[4]^2 - 2a[3]a[5])y + a[5]^2 = 0$$

The roots of the new equation are the squares of the old equation. Here is a computer program which generates the new array of coefficients. The input array is a, the new array is newa.

```
INITIALIZE ARRAY newa[n]
FOR i=1 TO n
FOR j=1 TO n
  LET k= (i+j)/2
  IF k is an integer then:
     newa[k] = newa[k] + a[i] * a[j] * (-1)^(i*j+1)
NEXT j
NEXT i
```

After repeating the root squaring process m times, the roots are appropriately separated. The coefficients of the final transformed equation can be broken up into linear or quadratic fragments. The "Numerical Mathematical Analysis" book describes a procedure of examining the pattern of signs of the coefficients to determine which roots are real (i.e. linear fragments) or complex (quadratic fragments). It is usually difficult to get a computer to understand patterns, so I took a different approach. The program takes each trio of coefficients and applies the b^2-4ac discrimenant test. If the discriminent was negative, the program assumes a pair of complex roots (and advances two units to get the next trio). If the discriment was positive, the program assumes one real root (and advances one unit to examine the next trio). To find a real root, take the 2^mth root of the ratios of the coefficients (the root squaring process was repeated m times).

But we are getting ahead of ourselves. The program uses the Brodetsky and Smeal improvement of the root squaring method. We use a different formula for obtaining the roots.

The Brodetsky and Smeal improvement uses an additional array for the h coefficients. The original equation is transformed by a small distance h from the origin. All h squared terms in any computation are ignored (h is vanishingly small). The terms of the h coefficients are stored in the array b. The starting array is the derivative of the polynomial. That is:

b[1] = 0

b[2] = n * a[1]

b[3] = (n-1) * a[2], etc.

Here is a computer program which generates the new array of b coefficients. Both the arrays a and b are used in the calculation. The new array is newb.

```
INITIALIZE ARRAY newb[n]
FOR i=1 TO n
FOR j=1 TO n
  LET k= (i+j)/2
  IF k is an integer then:
     newb[k] = newa[k] + a[i] * b[j] * (-1)^(i*j+1)
```

NEXT j
NEXT i

In the program STP which calculates does one step of the root squaring process, there are no actual arrays newa and newb. These were included in the above programs to make them easier to read. Program STP uses the stack to create the new arrays.

After going through this process several times, it is time to calculate the roots. The process is done with program FRTSQ. First, FRTSQ uses the discriminant test to divide the equation into linear and quadratic fragments. A linear fragment is given as follows:
$(a[i-1] + b[i-1]h)x^q + (a[i] + b[i]h) = 0$; where $q=2^m$

A little rearanging gives:

$$x = \frac{2^m}{(b[i-1]/a[i-1] - b[i]/a[i])}$$

Notice that using this formula, we obtain the correct sign of the root without any trial and error.

A quadratic fragment is given as follows:
$(a[i-1] + b[i-1]h)y^2 + (a[i] + b[i]h)y + (a[i+1] + b[i+1]h) = 0$
where $y = (x-h)^m$

After a fair amount of manipulations (see Numerical Mathematical Analysis), you obtain the magnitude of the complex roots (r) from:
$r^{2m} = a[i+1]/a[i-1]$
The complex roots can be written u+iv and u-iv where:
$u = (b[i-1]/a[i-1] - b[i+1]/a[i+1])\ (r^2/2^{m+1})$
and $v = \mathrm{SQR}\ (r^2 - u^2)$

Of course, these are just estimated values of the roots. To obtain better values, we use the Newton-Raphson method. The method geometrically approaches a root by setting $z = z_o - f(z_o)/f'(z_o)$

After getting a better value for z, the process can be repeated. This method works for real and complex roots. The program repeats the root correction up to 20 times. Experience has shown that in some situations, the convergence to the correct root is slow.

Why the Program Fails, and How to Cope

When the root squaring process has not gone enough steps, then the program cannot hope to give the right answers. Sometimes the program returns doubled roots when there are no doubled roots in reality. Sometimes it returns unstable values (the Newton-Raphson corrections swing back and forth). There may be other patterns, but I have not noticed them. I have supplied an additional program GDRT. The program (which also calls program STBL) takes a list of "roots" an throws away the duplicates and the unstable ones. The remaining roots are used to create a new polynomial as follows:
(x-r[1]) (x-r[2]) (x-r[3]) ...

This polynomial is divided into the original polynomial that we want to solve. This new polynomial, of lesser degree than the original, (called the reduced equaiton) is solved using MROOT. The roots of it and the "good roots" are placed on a combined list. A new score is generated to see if we can trust the revised list of roots.

Required Programs

It is assumed that you will only want root squaring on fifth degree equations and above. For lower degree, it uses QUD, CUBIC, or QUAR found in the HP book Mathematical Applications. These programs give closed form solutions for these equations.

Required programs: QUD, CUBIC, QUAR, PMULT, PDIV, and PVAL

I have made some minor changes to some of these programs. The programs QUD, CUBIC, and QUAR have been modified so that they return all the roots in list form in level 1.

At the end of program QUD, add the following: 2 ->LIST

At the end of program CUBIC, add the following: 3 ->LIST

Program QUAR has more extensive changes. QUAR has a reference to CUBIC. Omit the 3 ->LIST after it (this is now done by CUBIC). There are two references to QUD. Add LIST-> DROP after these. At the very end, add the following: 4 ->LIST

The program uses several HP-28S programs: MROOT, RTSQ, STP, T100, TWEEK, FRTSQ, SCORE GDRT, and STBL. MROOT is the master program, a replacement for PROOT. It decides whether to use the roots squaring process for polynomials of degree 5 or above, or to use one of the HP programs for quadratic, cubic, or quartic equations. Program RTSQ starts the root squaring process. The program STP executes one step of the root squaring process (with the Brodetsky and Smeal improvement). The program T100 tests whether the coefficients are getting too close to the 10^{500} limit of the calculator. FRTSQ takes the final coefficients and calculates the rough roots (both real and complex). TWEEK executes the Newton-Raphson method a number of times to improve the rough calculation. Program SCORE calculates the ratio of the product of the roots and the last coefficient. If all the roots are correct, this value should be 1.

When you execute MROOT, level one of the stack should contain the coefficients of the polynomial to solve in list format. When the program finishes several minutes later, level one contains the score, and the second level of the stack contains the roots in list format.

Additional programs GTRT and STBL are used in the event the score shows the program did not return the correct roots. To use GTRT, put the list of roots, good and bad (as returned by MROOT et. al.). It returns a new score and a revised set of roots.

Software Listings

MROOT Program

Input Level 1: coefficients of polynomial in list form
Output Level 2: [if degree > 4] list of roots
Output Level 1: [if degree > 4] score in string form
Output Level 1: [if degree <= 4] list of roots

```
<<
DUP IF SIZE 1 > THEN          if 0 or 1 elements in list, return nothing
  DUP IF SIZE 5 > THEN        if over 5 elements, use root square
    12 RTSQ 450 .25 BEEP      ask for 12 steps, beep when finished
    ELSE LIST-> ->ARRY        old PROOT for degree 1-4
      DUP 1 GET /             divide by first coefficient
      ARRY-> LIST-> DROP      level 1 has degree plus 1
      1 - DUP 2 + ROLL DROP   discard leading coefficient
      {<<NEG 1 ->LIST>>       a list of programs to
      QUD CUBIC QUAR}         evaluate for each degree
      SWAP GET EVAL           evaluate the right one!
    END
  ELSE DROP { }               if 0 or 1 elements in list, return empty list
END
>>
```

RTSQ Program

Input Level 2: coefficients in list form
Input Level 1: 12
Output: see FRTSQ Program

```
<<
SWAP LIST-> ->ARRY RE         save equation in array (real only!)
DUP 1 GET /                   normalize (divide by first coefficient)
ARRY-> LIST-> DROP ->LIST     put back into list form
DUP SIZE -> nt a n            nt is the number of times to repeat root square
                              a is the list of coefficients
                              n is the number of terms
<< { n 1 - } 0 CON            blank array for derivative
ARRY-> LIST-> DROP ->LIST     put into list form
1 n 1 - FOR i
  a i GET n i - * SWAP PUT    calculate derivative
NEXT 'DPNEQ' STO              save derivative in DPNEQ
a 'PNEQ' STO                  store polynomial in PNEQ
a 0 DPNEQ LIST-> 1 + ->LIST   use the derivative for h coefficients
SWAP                          level 1 has regular; level 2 has h coefficients
1 nt FOR i
  STP IF T100 THEN            do the root square, test size of coefficients
    i 'nt' STO 200 'i' STO    save revised nt; stop FOR NEXT loop
  END
NEXT
nt FRTSQ >>                   finish up calucation
>>
```

STP Program

Input Level 2: old B list (list of h coefficients)
Input Level 1: old A list (list of root square coefficients)
Output Level 2: new B list
Output Level 1: new A list

```
<<
DUP SIZE 0
-> b a n k <<                    b = h coefficient list;
                                 a = coefficient list
                                 n = number of terms (highest power plus 1)
                                 k = temporary value
{ n } 0 CON
ARRY-> LIST-> DROP ->LIST DUP    two new lists for the new coefficients
1 n FOR i                        outer loop for new a
  1 n FOR j                      inner loop for new a
    i j + 2 / 'k' STO            (i+j)/2 is used often; save as k
    IF k FP 0 == THEN            test if k is even
      DUP k GET a i GET
      a j GET * -1 i j *
      1 + ^ * + k SWAP PUT       save new value for a list
    END
  NEXT
NEXT SWAP                        entire new set of a coefficients; SWAP for b
1 n FOR i                        outer loop for new b
  1 n FOR j                      inner loop for new b
    i j + 2 / 'k' STO            (i+j)/2 is used often; save as k
    IF k FP 0 == THEN            test if k is even
      DUP k GET a i GET
      b j GET 2 * * -1 i j *
      1 + ^ * + k SWAP PUT       save new value for b list
    END
  NEXT
NEXT                             entire new set of b coefficients
SWAP >>                          a in level 1; b in level 2
>>
```

T100 Program

Input Level 1: A list of coefficients
Output Level 2: unchanged A list
Output Level 1: true if one or more values exceed 1E100

```
<<
DUP SIZE -> a n <<               standard a and n
a 0 1 n FOR i                    for each element
  a i GET ABS 1 + LOG
  ABS 100 > +                    check if over 1E100
NEXT
>>
```

FRTSQ Program

Input Level 2: final B array
Input Level 1: final A array
Output Level 2: roots in list form
Output Level 1: score (based on product of roots)

```
<<
3 ROLLD DUP
SIZE 0 0
-> m b a n r u <<              m is actual # of times we did the root square
                               b is the epsilon coefficient array
                               a is the regular root square array
                               n is number of coefficients
                               r is a temporary
                               u is a temporary
2 n FOR i                      loop through the roots
  IF i n == THEN 1             report true for last root
  ELSE a i GET DUP *
    4 a i 1 - GET              calculating discriminant
    a i 1 + GET * * >          report true for real; false for complex
  END
  IF THEN                      real root
    2 m ^ b i 1 - GET
    a i 1 - GET /
    b i GET
    a i GET / - / TWEEK        calculate real root and improve
  ELSE                         else a complex root
    a i 1 + GET
    a i 1 - GET / ABS
    2 m 1 + ^ INV ^ 'r' STO    r is the radius of the complex root
    b i 1 - GET
    a i 1 - GET /
    b i 1 + GET
    a i 1 + GET / - r DUP
    * * 2 m 1 + ^ / 'u' STO    u is the real portion of the root
    u r DUP * u DUP *
    - ABS √ R->C TWEEK         calculate and improve complex root
    DUP CONJ                   conjugate complex root
    i 1 + 'i' STO              bump i by 1 since we have two roots
  END
NEXT
n 1 - ->LIST                   put roots into list form
SCORE                          obtain the accuracy score
```

TWEEK Program

Input Level 1: raw unimproved root
Output Level 1: root improved by many repetitions of the Newton-Raphson method

```
<<
1 20 FOR i                         repeat up to 20 times
  DUP DUP DUP PNEQ SWAP PVAL       get function value at "root"
  DPNEQ ROT PVAL                   get derivative at "root"
  IF DUP 0 == THEN DROP 0          if derivative is zero, don't change root
  ELSE /                           ratio; function divided by derivative
  END -                            improved root value; root minus ratio
  DUP ROT - R-P RE                 find how much root has changed
  IF 0 == THEN i 18 MAX            if unchanged, advance the loop counter
  'i' STO END
NEXT                               loop through more times
>>
```

SCORE Program

Input Level 1: roots in list format
Output Level 2: unchanged list of roots
Output Level 1: score in string format

```
DUP LIST-> -> n                    n has the number of roots
<< 1 n 1 - START * NEXT            multiply all the roots
PNEQ n 1 + GET                     get last coefficient
DUP IF 0 == THEN DROP "0 "         if last coefficient is zero, don't divide
  ELSE / NEG "1 "                  if it is non-zer, get the ratio
END SWAP ->STR + >>                place on the stack the score as a string
>>
```

GDRT Program

Input Level 1: List of roots in list form; some are incorrect
Output Level 2: List of roots in list form, hopefully all correct
Output Level 1: score in string form

```
<< DUP SIZE -> r n              r is the list of roots
                                n is the number of roots
<< 0 'k' STO IF n 1 > THEN      init k; look for duplicate roots if n > 1
  PNEQ 1 n 1 - FOR i            put poly on stack
    0 'v' STO
    1 + n FOR j                 looping through the roots
      IF r i GET r j GET        compare ith and jth root to see if equal
      - R->P RE 1E-10 < THEN    if two roots are virtually identical
        1 'v' STO END           set a marker that indicates a duplicate
    NEXT IF v 0 == THEN         if not a duplicate
    r i GET STBL END            STBL puts the unique root on the "good" list
  NEXT 'v' PURGE                finish comparing all the combinations
END r n GET STBL                throw the last one on the list as well
k ->LIST DUP LIST-> DROP { 1 }  put good roots into the stack
1 k START SWAP                  loop through the good roots
  NEG 1 SWAP 2 ->LIST PMULT     create polynomial of good roots
NEXT PNEQ SWAP PDIV DROP DROP   divide into original equation
'k' PURGE MROOT                 find roots of reduced equation
DUP IF TYPE 2 == THEN DROP      if top is string (the score) then drop it
END + SWAP 'PNEQ' STO SCORE >>  add the new roots to the previous list
>>                              then score the combined list
```

STBL Program

Input Level 1: possible good root
Output Level 1: good root if it is stable, otherwise nothing

```
<< DUP DUP PNEQ SWAP PVAL
DPNEQ ROT PVAL /                calculate root correction
R->P RE IF .000001 < THEN       if the root is stable
  k 1 + 'k' STO                 remember it
  ELSE DROP                     if unstable, forget about it
END
>>
```

Some Examples

One particularly difficult equation is:

$x^8 + 7.73x^7 + 12.84x^6 - 1.111x^5 - 55.7x^4 - 125.3x^3 - 157.9x^2 - 112.3x - 56.3 = 0$

To solve, place the list of coefficients in a list:

{1 7.73 12.84 -1.111 -55.7 -125.3 -157.9 -112.3 -56.3 }

After about 4 minutes, we get the answer. The score is 1, so we can trust the results. The roots are:

-.275981481707	+-	.957681452645	i
-.569421123577	+-	1.33932524678	i
-1.29450777126	+-	.715583217023	i
2.17457247033			
-5.62475171764			

This equation is nasty for the root squaring process because it has three pairs of complex roots, and because the magnitude of two of the pairs vary by a ratio of only 1.02. The program, with its use of the Brodetsky and Smeal improvement is not confused by the number of complex roots. The repeated use of the Newton-Raphson method yields accurate results despite the slow convergence of the roots. While four mininutes is a long time to solve a problem, doing this manually (with only a few decimal places) would take days. Scarborough uses this equation as an exercise, but gives the wrong solution in the end of the book!

Here is a 20 degree equation in list form:

{1 0 -3 2.5 8 -12 5 8 24 -30 0 0 45 -60 157.2 -52 41 42 4 -2.5 2}

After we run MROOT, we get a very bad score (.001604). What has happened is that the program accidientially repeated one pair of complex roots. In addition, it reported 6 real roots when actually there are no real roots at all. Press DROP to get rid of the score and bring the list of roots to level 1. Press GDRT to try again. It eliminates the 8 bad roots, creates a 12 degree equation based on the 12 good roots, and divides into the original polynomial to get an 8th degree polynomial. The reduced equation is solved (correctly). The revised list of roots and the revised score is returned by the program. The new score is .999999999945, which is close enough to 1. This means we can trust the 10 pairs of complex roots. It is quite impressive that the program was able to deal with 10 pairs of complex roots. Total execution time was about 45 minutes. Here is the correct list of roots:

-1.80777576529	+-	.771777519729	i
1.34801982239	+-	.79552493042	i
.075056314795	+-	1.01360455599	i
.450179787303	+-	.742890160974	i
-.045129877568	+-	.206581781691	i
.184783179144	+-	.251367269368	i
.703544043027	+-	1.33785609926	i
1.30962750061	+-	.455669938931	i
-1.183335801	+-	.624646075475	i
-.674963520093	+-	1.12090106015	i